FLASH: A Framework for Programming Distributed Graph Processing Algorithms

Xue Li[1], Ke Meng[1], Lu Qin[2], Longbin Lai[1], Wenyuan Yu[1], Zhengping Qian[1], Xuemin Lin[3], Jingren Zhou[1]
[1]Alibaba Group
[2]Centre for Artificial Intelligence, University of Technology, Sydney, Australia
[3]Antai College of Economics & Management, Shanghai Jiao Tong University
[1]{youli.lx, mengke.mk, longbin.lailb, wenyuan.ywy, zhengping.qzp, jingren.zhou}@alibaba-inc.com
[2]lu.qin@uts.edu.au [3]xuemin.lin@gmail.com

***Abstract*—As a result of decades of studies, a broad spectrum of graph algorithms have been developed for graph analytics, including clustering, centrality, traversal, matching, mining, etc. However, the majority of recent graph processing frameworks only focus on a handful of fix-point graph algorithms such as breadth-first search, PageRank, shortest path, etc. It leaves the distributed computation of a large variety of graph algorithms suffering from low efficiency, limited expressiveness, or high implementation complexity with existing frameworks.**

In this paper, we propose FLASH, a framework for programming distributed graph processing algorithms, which achieves good expressiveness, productivity and efficiency at the same time. Thanks to its high-level interface, FLASH allows users to implement complex distributed graph algorithms with high performance with only a few lines of code. We have implemented 72 graph algorithms for 49 different problems in FLASH. In further evaluations, we found that FLASH beats other state-of-the-art graph processing frameworks with the speedups of up to 2 orders of magnitudes while takes up to 92% less lines of code.

***Index Terms*—Graph computing, Programming model, Distributed computing, Code generation**

I. INTRODUCTION

Graph algorithms serve as the essential building blocks for a diverse variety of real-world applications such as social network analytics [2], data mining [3], network routing [4], and scientific computing [5]. As the graph data becomes increasingly huge, there is an urgent need of scaling graph algorithms in the distributed context, and many distributed graph frameworks have emerged to fill in the gap, such as Pregel [6], Giraph [7], GraphLab [8], PowerGraph [9], GraphX [10], Gemini [11] and others [12], [13], [14], [15], [16]. However, as pointed out by [17], all these graph processing frameworks are only evaluated via a handful of specific graph algorithms that are similar in computation patterns. Such evaluation is far from sufficient lacking in the coverage of diversity and usability required from real-world applications. To comprehensively evaluate a distributed graph framework, we carefully consider three metrics that we call **EPE**, namely *expressiveness*, *productivity* and *efficiency*. Specifically,

- Expressiveness represents the capability of the programming interface to express different kinds of graph algorithms. Expressiveness is arguably the most important metric that must be fulfilled in order to support the diverse graph applications in practice.
- Productivity shows the convenience of the interface or the required effort for users to implement their algorithms, which is also important given that otherwise graph computation will be a privilege to a few expertise [16].
- Efficiency refers to the performance factor of executing graph algorithms, which should be concerned because real-world applications on large-scale graphs are often time-consuming and/or memory-intensive.

After thoroughly analyzing existing graph processing frameworks, we find out that none of them has arrived at the proper tradeoff for all the EPE metrics. To pursuit productivity, an abstraction called "thinking like a vertex" (or vertex-centric) was proposed in Pregel [6], and shared among many existing graph processing frameworks. Under this philosophy, graphs are divided into partitions for scalability and updates are bound to the granularity of vertices for parallelization. The vertex-centric implementation of a graph algorithm follows a common iterative, single-phased and value-propagation-based (short of *ISVP*) pattern [17]: the algorithm runs iteratively until convergence, and in each iteration, all vertices receive messages from their neighbors to update their own states, then they send the updated states as messages to the neighbors for the next iteration. Due to the productivity of the vertex-centric model, a lot of graph frameworks follow the philosophy for their abstraction, including the GraphLab variant [8], the Scatter-Gather model [18], and the *GAS* model [9].

Such high-level abstraction brings productivity to some extent to users, however, at the sacrifice of expressiveness and efficiency. Regarding expressiveness, we argue that the abstraction, while designed specifically for the ISVP algorithms, is almost infeasible to be applied to a large variety of algorithms that are not of the kind, such as K-clique Counting, Rectangle Counting, Betweenness Centrality and the optimized Connected Components algorithm [19], to just name a few. Actually, there are typically tens of algorithms available to representative graph processing frameworks. As a comparison, NetworkX [20], a Python graph library, supports over 328 graph algorithms. At the same time, modern graph scenarios bring in the needs of more advanced and complex graph algorithms, which poses a big challenge for existing graph processing frameworks. After investigating representa-

tive distributed graph algorithms, including many non-ISVP ones, we have distilled three requirements that are critical for programming them efficiently and productively in a distributed context, namely (R1) flexible control flow; (R2) operations on vertex sets; and (R3) beyond-neighborhood communication. As examples, Table I lists what requirements are needed by the evaluated algorithms (more algorithms can be found in [1]). However, existing graph frameworks all fall short in meeting these requirements. For example, Pregel does not offer R1 and R2, while GAS fails to support all three requirements. Ligra [21], [22] is the programming model that has fulfilled the most requirements so far (R1 and R2). However, Ligra still has limitations for expressing non-ISVP algorithms because of the absence of R3. In addition, it is built upon a shared-memory architecture, thus not suitable for programming graph algorithms in a distributed context.

Regarding efficiency, it's not surprising that some hard-coded algorithms can run much faster than the high-level alternative implementation on a framework. The general graph frameworks, especially distributed frameworks lose efficiency because of communication/synchronization overhead, load balance issues and higher software complexities [11]. As we evaluated, a hardcoded algorithm for Connected Components [19] can perform orders of magnitude faster than the vertex-centric counterparts. To overturn the efficiency issue, Gemini [11] has been developed that significantly lower the latency of executing certain algorithms. However, the user is often required to program more codes compared to the vertex-centric alternatives, which harms the productivity to some degree. Moreover, in order to exploit the extreme efficiency, some constraints are enforced by Gemini that can deteriorate the expressiveness: it does not support the above three critical requirements, and it requires the vertex property to be fixed-length and the core computation to satisfy the *associative* and *commutative* properties. Obviously, a graph framework based on the low-level interfaces such as MPI can provide the optimal result for expressiveness and efficiency, but it may not be appealing to the users in terms of productivity.

Motivated by this, we propose a new distributed graph processing framework called FLASH in this paper, and claim that FLASH finally approaches a sweet spot for all EPE metrics. We base the FLASH programming model on Ligra to inherit its support for the requirements of flexible control flow and operations on vertex sets. By further enabling beyond-neighborhood communication, FLASH improves the expressiveness for programming a diverse variety of distributed graph algorithms. Note that Ligra is a single-machine parallel library, and our extension of Ligra to the distributed context is non-trivial, for which we must carefully handle communication, synchronization, data races and task scheduling. To do so, we propose a middleware called FLASHWARE that hides all the above details for distribution, and provides the capability to apply multiple system optimizations automatically and adaptively at the runtime. In fact, we have implemented 72 graph algorithms in FLASH for 49 different commonly used applications. Among them, some algorithms are extremely difficult or nearly impossible to implement on existing graph processing frameworks, including the optimized algorithm to compute minimum spanning tree [19], faster betweenness centrality algorithm [23], k-core decomposition [24], graph coloring [25], k-clique counting [26], to just name a few. Moreover, thanks to FLASHWARE, we can now program much more succinct codes using the FLASH programming interfaces, which also helps productivity. Table I summarizes the comparison in expressiveness and productivity of FLASH and the representative graph frameworks, while Figure 1 presents the comparison in efficiency. No a single framework can beat FLASH in all metrics, while FLASH does outperform Pregel+ and PowerGraph by a large margin in all aspects.

TABLE I: Comparison of different models.

Algo. [20]	**Pregel**	**GAS**	**Gemini**	**Ligra**	**FLASH**
Expressiveness & Productivity [LLoCs [27], lower is better]					
CC-basic	● 30	● 36	● 50	● 26	● 12
CC-opt[13]	◒ 63	○	○	○	● 56
BFS	● 22	● 25	● 56	● 20	● 13
BC[12]	◒ 49	◒ 162	◒ 139	● 75	● 33
MIS[2]	● 48	● 53	● 112	● 37	● 23
MM-basic	● 57	● 66	● 98	● 59	● 20
MM-opt[13]	◒ 84	○	○	○	● 27
KC[1]	◒ 35	◒ 32	○	● 45	● 20
TC	● 31	◒ 181	○	● 38	● 22
GC	● 48	● 58	○	○	● 24
SCC[12]	◒ 275	○	○	○	● 74
BCC[123]	◒ 1057	○	○	○	● 77
LPA	● 51	● 46	○	○	● 26
MSF[13]	◒ 208	○	○	○	● 24

The algorithms are demonstrated in the full version [1].
The three critical requirements are: flexible control flow[1], operations on vertex sets[2] and beyond-neighborhood communication[3].
● means that it is well-supported; ○ means that we failed to express it; ◒ means that it could be implemented in a non-intuitively way at the cost of performance.

In summary, we make the following contributions.

(1). We propose the FLASH model, a novel high-level abstraction for programming distributed graph algorithms (Section III). We define its succinct interface after presenting some preliminaries (Section II), and show its expressiveness through some representative examples. We also make a comparison with other models to highlight FLASH's advantages.

(2). We provide an efficient implementation of FLASH (Section IV) based on a novel design of the system architecture and a middleware named FLASHWARE. Moreover, we explore adaptive runtime choices to further enhance the performance.

(3). We provide a thorough experimental evaluation of FLASH considering the EPE metrics (Section V). The results demonstrate FLASH's capability of expressing many advanced algorithms, while providing a satisfactory performance at the same time. It beats other state-of-the-art frameworks in 84.5% cases, and may achieve significant speedups of up to 2 orders of magnitudes while takes up to 92% less lines of code.

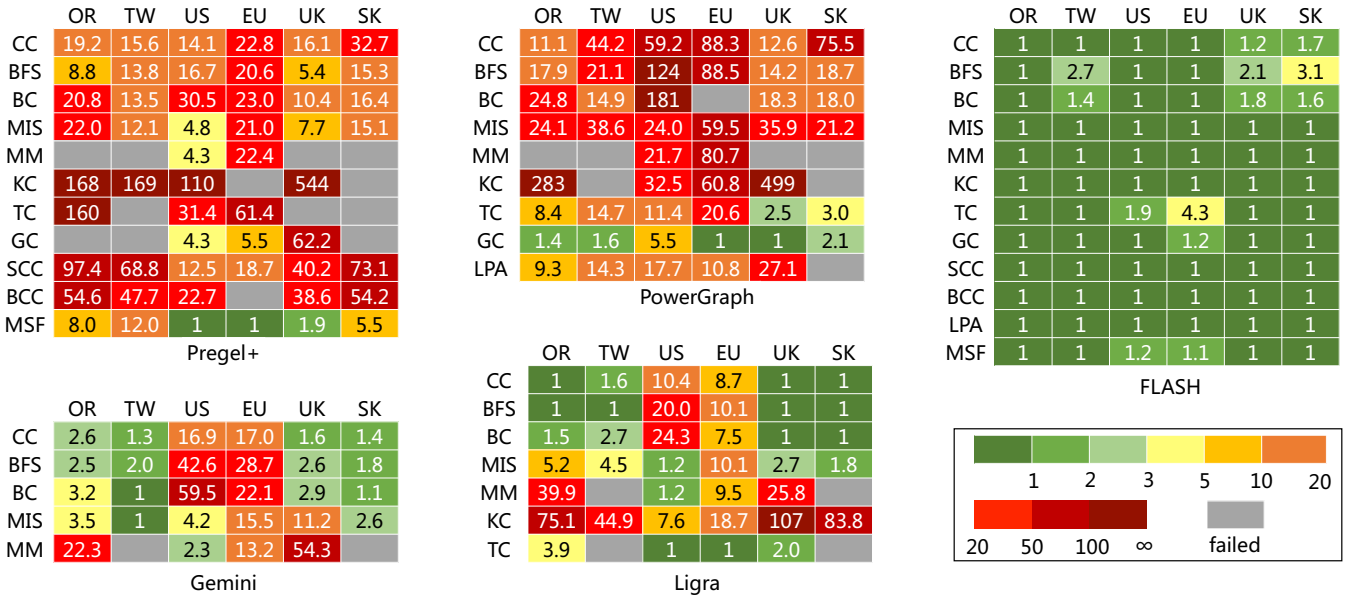

Fig. 1: A heat map of slowdowns of various frameworks compared to the fastest framework for 12 representative applications on 6 real-world graphs. "failed" means that the execution did not terminate within 5000s or failed due to exhausted memory.

II. Preliminaries and Related Work

We first define some basic graph notations and discuss the mainstream programming models of existing frameworks.

Functions. FLASH uses the functional programming paradigm. A variable with a type is denoted as $var : type$. A function $f(x : X) \mapsto Y$ means that for each input $x \in X$, it has an output $y \in Y$ such that $f(x) = y$.

Graphs. FLASH is defined over the property graph, which can be represented as $G = (V, E)$. V represents a finite set of vertices, and each $v \in V$ has a unique identifier ($v.id \in \mathbb{N}$) and some associated properties. With the cartesian product of sets A and B denoted by $A \times B$ where $A \times B = \{(a, b) : a \in A \wedge b \in B\}$, $E \subseteq V \times V$ represents a set of (directed) edges. For each edge (s, d), the first vertex s is the source of it and the second vertex d is the target. The number of vertices in a graph is denoted by $|V|$ and the number of edges is denoted by $|E|$. We denote a weighted graph by $G = (V, E, w)$, where w is a function which maps an edge to a real value, thus each edge $e \in E$ is associated with the weight $w(e)$.

Graphs partitions. In a distributed cluster containing m workers, the input graph will be partitioned into m partitions (also called subgraphs), with each worker holding one of the partitions ($P_i = (V_i, E_i)$ for worker i). A m-way partition scheme for the graph $G(V, E)$ should guarantee that $V = \cup_{i=1}^{m} V_i$ and $E = \cup_{i=1}^{m} E_i$. There are two main schemes for partitioning the graph, namely *edge-cut* and *vertex-cut*, and we use *edge-cut* which is more common in existing works. To be specific, each vertex belongs to only one partition, so that $V_i \cap V_j = \emptyset$, for $i \neq j$. Two endpoints of an edge may belong to different partitions. To this end, FLASH uses the master-mirror notion as PowerGraph [9] proposed: each vertex is assigned to and owned by one partition, where it is a *master*, as its primary copy. While in other partitions, there may also be replicas for this vertex, called *mirrors*. We will explain this more specifically in Section IV.

Graph algorithms. A graph algorithm takes a graph G as input, does processing and analytics on it to solve real-world problems. As assumed in many works, a graph algorithm updates the information stored in vertices, while edges are viewed as immutable objects. Generally, A graph algorithm propagates updates along the edges (original edges of G, or virtual edges generated dynamically during the execution) iteratively, until the convergence condition is met or a given number of iterations are completed. Vertices with ongoing updates are called *active vertices* (or *frontiers*), with outgoing edges from them called *active edges*. This paper considers the Bulk Synchronous Parallel (BSP) model [28], where an algorithm consists of a series of supersteps. In each superstep, computation and communication (message passing) take place on active vertices. The supersteps are executed synchronously, so that messages sent during one superstep are guaranteed to be delivered in the beginning of the next superstep.

Pregel. There are many parallel programming abstractions for processing large graphs [29], [30], [17]. Pregel [6] proposed a vertex-centric abstraction which is designed based on the BSP model. By distributing vertices and associated adjacent edges to each worker, the Pregel framework has fixed routines for all the algorithms. In each superstep, a Pregel program calls the `compute()` on each vertex, which performs the pre-defined user-specific computation. During the `compute()`, each vertex processes the incoming messages from the previous superstep, then sends messages to other vertices. The communication is based on message-passing, with messages can be aggregated if the user provides a `combine()` function. The early-aggregation of messages can reduce the number of messages to be materialized, thus improves the performance

and reduces the memory usage. The Pregel model greatly simplifies the parallel graph algorithms. Therefore, it almost becomes a standard of large graph processing, followed by many graph processing frameworks [7], [13], [12], [10], [11].

However, the Pregel model only exposes a single vertex to the users, and thus lacks in the global perspective, for example, to maintain a group of specific vertices. Additionally, the loop control flow is *constrained* by the model, making it only suitable for single-phased algorithms. As an example, the Betweenness Centrality algorithm [23] requires both flexible control flow (e.g. recursive loop) and operating on frontiers as vertex sets in each step, which cannot be easily expressed by this model. Actually, the implementation of it provided by [11] needs more than 400 lines of code in total. Following this vertex-centric philosophy, FLASH makes an improvement on the expressiveness by allowing the users to define flexible control flow and materialize vertex sets to operate upon.

GAS. GraphLab [8] and PowerGraph [9] are also based on the vertex-centric abstraction. They proposed the "Gather-Apply-Scatter" (GAS) model to further simplify the graph algorithms, but at the cost of limiting the data-exchange to only happen between adjacent vertices. GAS hides the communication details from programmers, and the users only have the view of each vertex and its neighbors, which means that the control flow of a graph algorithm is highly rigid. Thus, GAS is less expressive but more effortless than Pregel. In addition to supporting flexible control flow and operating on arbitrary vertex sets, FLASH removes the between-neighborhood limitation of GAS to support the communication beyond neighbors.

Ligra. Ligra [21] proposed a new model that supports a ***vertexSubset*** type. It represents a subset of vertices $U \subseteq V$ of the graph G. Based on this data structure, it is easy to apply the update logics on any subset of vertices every time. It exposes the control flow to the users, supporting arbitrary combination of these operations during the execution. Ligra is proved to be useful when programming a wide variety of parallel graph algorithms in the shared-memory environment.

Nevertheless, Ligra is still limited in some aspects. Since the communication is through the EDGEMAP interface, only the messages along the edges are possible. However, communication beyond neighborhood or along a set of specific edges is an important feature in some advanced graph algorithms [19]. And Ligra is designed for shared-memory systems, thus lacks the distributed semantics. It requires the programmers to use low-level instructions (e.g., the compare-and-swap instruction), which are not fitted in the distributed scenario. By extending Ligra's model, FLASH is more expressive that supports communicating beyond neighbors. More importantly, FLASH does not depend on the shared-memory architecture, making it suitable for distributed processing.

III. PROGRAMMING MODEL

As depicted in Section II, previous frameworks fail to achieve the **EPE** metrics due to fixed control flow, lack of view on arbitrary vertex subsets, neighborhood-exchange limitation and the dependence on the shared-memory architecture. To address the challenges, we propose the FLASH programming model by making a sufficient extension to Ligra since Ligra addressed the first two challenges. In this section, we first show the interface of FLASH, and how to flexibly express graph algorithms with the given primitives. Then we show that our FLASH model can be fully compatible with the well-known vertex-centric models. Finally, we highlight some characteristics of FLASH to show its advantages.

A. Interface

FLASH is a functional programming model specific for distributed graph processing. It follows the Bulk Synchronous Parallel (BSP) computing paradigm [28], with each of the primary functions (SIZE, VERTEXMAP and EDGEMAP) constitutes a single superstep. We made the interface of FLASH much similar to Ligra's, therefore, it is easy to port a program written in Ligra to our model. The *vertexSubset* type represents a set of vertices of the graph G, which only contains a set of integers, representing the vertex id for each vertex in this set. The associated properties of vertices are maintained only once for a graph, shared by all *vertexSubset*s. The following describes the APIs of FLASH based on this type.

(1). $\textbf{SIZE}(U : vertexSubset) \mapsto \mathbb{N}$

This function returns the size of a *vertexSubset*, i.e., $|U|$.

(2). $\textbf{VERTEXMAP}(U : vertexSubset,$
$F(v : vertex) \mapsto bool,$
$M(v : vertex) \mapsto vertex) \mapsto vertexSubset$

The VERTEXMAP interface applies the map function M to each vertex in U that passes the condition checking function F. The ids of the output vertices form the resulting *vertexSubset*. That is to say, we have:

$Out = \{v.id | v.id \in U \wedge F(v) = true\}$
$v^{new} = M(v),\ v.id \in U \wedge F(v) = true$

This function is used to conduct local updates. Specially, the M function could be omitted for implementing the *filter* semantics, with the vertex data unchanged. The execution of VERTEXMAP on each vertex is independent, thus it can run in parallel naturally, as shown in Algorithm 1.

(3). $\textbf{EDGEMAP}(U : vertexSubset,$
$H : edgeSet,$
$F(s : vertex, d : vertex) \mapsto bool,$
$M(s : vertex, d : vertex) \mapsto vertex,$
$C(v : vertex) \mapsto bool,$
$R(t : veretex, d : vertex) \mapsto vertex) \mapsto vertexSubset$

The EDGEMAP interface in Ligra is used to transfer messages between neighbor vertices. We extend and redefine this interface for stronger expressiveness and the support for distributed semantics. For a graph $G = (V, E)$, EDGEMAP

Algorithm 1 VERTEXMAP

```
1: function VERTEXMAP(U, F, M)
2:     Out = {}
3:     parfor u.id ∈ U do
4:         if (F(u) == true) then
5:             u^new = M(u)
6:             Add u.id to Out
7:     return Out
```

applies the update logic to the specific edges with source vertex in U and target vertex satisfying C. H represents the edge set to conduct updates, which is E in common cases. We allow the users to define arbitrary edge sets they want dynamically at runtime, even virtual edges generated during the algorithm's execution. The edge set can be defined through defining a function which maps a source vertex id to a set of ids of the targets. We also provide some pre-defined operators for convenience, such as reverse edges (`reverse(E)`), two-hop neighbors (`join(E, E)`), or edges with targets in U (`join(E, U)`). This extension makes the communication beyond the neighborhood-exchange limitation.

If a chosen edge passes the condition checking (F), the map function M is applied on it. The output of the function M represents a temporary new value of the target vertex. This new value is applied immediately and sequentially if it is in the *pull* mode, while in the *push* mode, another parameter R is required to apply all the temporary new values on a specific vertex to get its final value. It is unnecessary in the Ligra's API because it is a shared-memory framework, which uses atomic operations to ensure consistency. On the contrary, the FLASH model is designed for distributed systems, so we use a reduce function R, which takes an old value and a new value for a single vertex, and reduces them to output the updated state. The updated target vertices form the output set of EDGEMAP. The reduce function R should be associative and commutative to ensure correctness, or R is not required for sequentially applying M, i.e., to run EDGEMAP always in the *pull* mode, as we will explained in Section III-C.

More precisely, the active edge set is defined as:

$E_a = \{(s, d) \in H | s.id \in U \wedge C(d) = true\}$.

Then, F and M are applied to each element in E_a. If it is in the *pull* mode:

$d^{new} = M(s, d^{new}), (s, d) \in E_a \wedge F(s, d^{new}) = true$.

Or, in the *push* mode:

$T = \{M(s, d) | (s, d) \in E_a \wedge F(s, d) = true\}$,

$d^{new} = R(..., R(T_2^d, R(T_1^d, d))), T_i^d \in T \wedge T_i^d.id = d.id$.

And the ids of the updated targets form output set:

$Out = \{d.id | (s, d) \in E_a \wedge F(s, d) = true\}$.

The function C is useful in algorithms where a value associated with a vertex only needs to be updated once. We retain the default function used by Ligra (CTURE) which always returns true, since the user does not need this functionality sometimes. Similarly, the F function of EDGEMAP and VERTEXMAP can also be supplied using CTURE, if it is unnecessary.

The auxiliary operators. Other auxiliary APIs are provided by FLASH for conveniently conducting set operations, including UNION, MINUS, INTERSACT, ADD, CONTAIN and so on.

B. Examples

To demonstrate the usage of FLASH and display its ability to express graph algorithms, we show two representative examples. Please refer to the full version [1] for more examples.

Breadth First Search (BFS). As with standard parallel BFS algorithms [31], [32], we implement BFS in FLASH, as shown in Algorithm 2. For each vertex, a property named dis is created and initialized to represent the distance from the root to this vertex. On each iteration/superstep i (starting from 0), the frontier U_i contains all vertices reachable from the root in i hops ($v.dis = i$, $\forall v \in U_i$). At the beginning of the algorithm, a *vertexSubset* that only contains the root is created, representing the frontier. To use a global variable such as r in a local function, we provide a $bind$ operator to supply additional input parameters (line 11). In each of the following supersteps, the EDGEMAP function is applied on outgoing edges of the frontier, to check if any neighbor d of an active vertex s is visited. If d has not been visited, updates it and then adds it to the next frontier. The COND function tells EDGEMAP to only consider the neighbors not been visited. Although it could be replaced by CTRUE, we provide it for efficiency. As dis for a vertex d is ensured to be same no matter it is updated by which neighbor in the same superstep, we can simply remain any new value for it in the REDUCE function. The iterative execution will terminate when there are no vertices in the frontier, means that all reachable vertices from the root have been visited.

Algorithm 2 BREADTH-FIRST SEARCH

```
1: function INIT(v, root):
2:     v.dis = (v.id == root ? 0 : INF)
3:     return v
4: function FILTER(v, root): return v.id == root
5: function UPDATE(s, d):
6:     d.dis = s.dis + 1
7:     return d
8: function COND(v): return v.dis == INF
9: function REDUCE(t, d): return t
10:
11: U = VERTEXMAP(V, CTRUE, INIT.bind(0))          ▷ initialize
12: U = VERTEXMAP(V, FILTER.bind(0))               ▷ root=0
13: while SIZE(U) ≠ 0 do
14:     U = EDGEMAP(U, E, CTRUE, UPDATE, COND, REDUCE)
```

Betweenness Centrality (BC). The betweenness centrality index [33] is useful in social network analysis and has been well studied. Precisely, the betweenness centrality of a vertex v, denoted by $C_B(v)$, is equal to $\sum_{s \neq v \neq t \in V} \delta_{st}(v)$. The pair-dependency $\delta_{st}(v)$ is calculated by $\frac{\sigma_{st}(v)}{\sigma_{st}}$, where σ_{st} is the number of shortest paths from s to t, and $\sigma_{st}(v)$ is the number of shortest paths from s to t that pass through v. To compute it, [23] proposed an optimized algorithm.

This algorithm works in two phases, the first phase uses a BFS-like procedure to calculate the number of shortest paths from r to each vertex, and the second phase computes the dependency scores through a backward propagation. Since the frontiers visited in every step of the first phase need to be tracked, it is difficult to directly implement this algorithm in a traditional vertex-centric model which does not supply a *vertexSubset* structure. In contrast, its implementation in FLASH is intuitively, as Algorithm 3 shows. In the second phase, the edges are need to point in the reverse direction, so we define the edge set for the EDGEMAP to be `reverse(E)`.

C. Advantages of FLASH

Expressing other programming models. FLASH has the ability to simulate the traditional vertex-centric models. So

Algorithm 3 BETWEENNESS CENTRALITY

```
1: function INIT(v, r):
2:     if (v.id == r) then v.level = 0, v.num = 1, v.b = 0
3:     else v.level = −1, v.num = 0, v.b = 0
4:     return v
5: function FILTER(v, r): return v.id == r
6:
7: function UPDATE1(s, d):
8:     d.num = d.num + s.num
9:     return d
10: function COND1(v): return v.level == −1
11: function R1(t, d):
12:     d.num = d.num + t.num
13:     return d
14:
15: function LOCAL(v, curLevel):
16:     v.level = curLevel
17:     return v
18:
19: function F2(s, d): return d.level == s.level − 1
20: function UPDATE2(s, d):
21:     d.b = d.b + (d.num / s.num) * (1 + s.b)
22:     return d
23: function R2(t, d):
24:     d.b = d.b + t.b
25:     return d
26:
27: function BC(S, curLevel)
28:     if (SIZE(S) == 0) then return
29:     A = EDGEMAP (S, E, CTRUE, UPDATE1, COND1, R1)
30:     A = VERTEXMAP (A, CTRUE, LOCAL.bind(curLevel))
31:     BC(A, curLevel + 1)
32:     A = EDGEMAP (S, reverse(E), F2, UPDATE2, CTRUE, R2)
33:
34: r = 0                                   ▷ choose a BFS root
35: U = VERTEXMAP(V, CTRUE, INIT.bind(r))   ▷ initialize
36: U = VERTEXMAP(V, FILTER.bind(r))        ▷ create the frontier
37: BC(U, 1)          ▷ calculate the betweenness centrality scores
```

it is possible to port existing vertex-centric programs in our model. Below we outline a proof (see the full version [1] for details). In each superstep of the vertex-centric model, all active vertices (called *frontiers*) execute the same user-defined vertex function in parallel, which receives a set of messages as input ($inbox$) and can produce one or more messages as output ($outbox$). At the end of a superstep, the runtime receives the messages from the $outbox$ of each vertex and computes the set of active vertices for the next superstep. The local computation in each superstep can be implemented in FLASH through the VERTEXMAP, which processes the $inbox$ to produce the updated value and the $outbox$ for each vertex, and a following EDGEMAP function adds a message to the $inbox$ of the target.

Support for the three critical requirements. Besides expressing existing vertex-centric algorithms, FLASH provides the possibility of expressing more advanced algorithms. It is the first distributed graph processing model that satisfies all of the three critical requirements for programming non-ISVP algorithms. (1) We allow the users to define the arbitrary control flow by combining the primitives, thus FLASH can naturally support multi-phased algorithms. In traditional vertex-centric models, these algorithms are supported in an awkward way since they only allow to provide a single user-defined function. (2) The *vertexSubset* structure supplements the perspective of a single vertex, allowing to conduct updates on arbitrary vertices. Multiple vertex sets can be maintained at the same time, they can even be defined in a recursive function. Without this feature, a framework has to start from the whole graph every time and pick up specific vertices. (3) FLASH makes an extension to Ligra by allowing the users to provide the arbitrary edge set they want to transfer messages, even when the edges do not exist in the original graph. Therefore, algorithms that contain communication beyond neighborhood can be expressed intuitively, such as the optimized CC algorithm [19].

Dual update propagation model. During graph processing, the type of an active set may be *dense* or *sparse*, typically determined by the size of active vertices and associated outgoing edges. Ligra dispatches different computation kernels for different types of the active set: the *push* mode for sparse active sets and the *pull* mode for dense active sets. FLASH follows this design and extend it to fit the distributed scenario, using an adaptive switching between these two modes.

The EDGEMAP function calls one of EDGEMAPDENSE and EDGEMAPSPARSE (Algorithm 4 - 6) according to the density, which implements the *pull* mode and the *push* mode, respectively. The users can set the threshold to decide if it is dense. EDGEMAPDENSE loops all vertices in the graph in parallel and for each vertex v, it sequentially applies F and M for its neighbors that are in U and the edges are in H, until C returns false. After that, its id will be added to the result. Since all updates are applied immediately, R is omitted.

On the contrary, EDGEMAPSPARSE loops all vertices in the active set U in parallel and for each vertex $u \in U$, it executes $F(u, ngh)$ and $M(u, ngh)$ in parallel to update its qualified neighbors. If a neighbor is updated, it will be added to Out. As a vertex may be updated by different neighbors at the same time in a single EDGEMAPSPARSE function, all the new values will be applied on the target vertex through the R function.

This auto-switch scheme is proved to be useful for real-world graphs and is also adopted by some other works [34], [35], [11]. Also, FLASH 's dual mode processing is optional: users may choose to execute in only one mode through calling EDGEMAPDENSE/EDGEMAPSPARSE, instead of EDGEMAP.

Algorithm 4 EDGEMAP

```
1: function EDGEMAP(U, H, F, M, C, R)
2:     if (the density of U > threshold) then
3:         return EDGEMAPDENSE (U, H, F, M, C)
4:     else
5:         return EDGEMAPSPARSE(U, H, F, M, C, R)
```

Algorithm 5 EDGEMAPDENSE

```
1: function EDGEMAPDENSE(U, H, F, M, C)
2:     Out = {}
3:     parfor v.id ∈ V do
4:         v^new = v
5:         for (ngh, v) ∈ H do
6:             if (C(v^new) == 0) then break
7:             if (ngh.id ∈ U and F(ngh, v^new) == 1) then
8:                 v^new = M(ngh, v^new)
9:                 Add v.id to Out
10:    return Out
```

Algorithm 6 EDGEMAPSPARSE

```
1: function EDGEMAPSPARSE(U, H, F, M, C, R)
2:     Out = {}, Tmp = {}
3:     parfor u.id ∈ U do
4:         parfor (u, ngh) ∈ H do
5:             if (C(ngh) == 1 and F(u, ngh) == 1) then
6:                 Add (M(u, ngh), ngh.id) to Tmp
7:                 Add ngh.id to Out
8:     parfor v.id ∈ Out do
9:         v^new = v
10:        for (t, v.id) ∈ Tmp do
11:            v^new = R(t, v^new)
12:    return Out
```

IV. System Design and Implementation

To realize the programming model above, we design and implement a new distributed framework. In this section, we first describe the system architecture. The technique details of the main components will also be presented. And then, we will introduce some system optimizations we have explored.

There are several main components in the framework, as shown in Figure 2. The first is a code generator which takes the high-level FLASH APIs as input, and generates execution code to be run on the second component named FLASHWARE, which is a middleware designed and optimized for the distributed graph processing. The FLASHWARE executes the code produced by the code generator on the distributed runtime, which contains multiple CPUs of the cluster machines. Each process acts as an individual worker and could contain multiple working threads. Each worker processes a fragment of the distributed computation, and the communication between different workers is implemented through MPI.

A. Graph Partition

The graph is partitioned using an *edge-cut* scheme, as we described in Section II. Every worker is assigned with a set of disjoint vertices (and the associated edges). For a worker, the vertex data is held in its memory. While for the edges, it depends: if there are enough memory capacity, the edges are cached in memory all the time; otherwise, they are only loaded from disks when necessary. The built-in partitioner is hash-based as it is most-frequently used, and it is more effective than other lightweight strategies in FLASH according to our evaluation (in the full version). FLASH also allows the users for specifying their own edge-cut partition strategy (e.g., METIS [36]) to pursuit even better workload balance. As this is not the focus of this work, we do not dive into more details.

Suppose an m-worker cluster, the given graph $G = (V, E)$ will be partitioned into m partitions. V_i is the vertex subset held by the i-th worker, with each vertex in V_i owns a *master* on this worker. There are also mirrors created for remote vertices (i.e., vertices assigned to other workers). Figure 2 gives an example, where the first three vertices locate at worker #1, and the other three vertices belong to worker #2. For each *vertexSubset*, a worker simply maintains a set of vertex ids, representing the master vertices in the set that locate on it.

B. FLASHWARE

FLASHWARE is designed as a middle layer that completes intra-node updating and inter-node communication (message passing). FLASHWARE enables FLASH's interface to hide the details of communication and data distribution, as well as provides the capability to apply multiple system optimizations automatically and adaptively at the runtime.

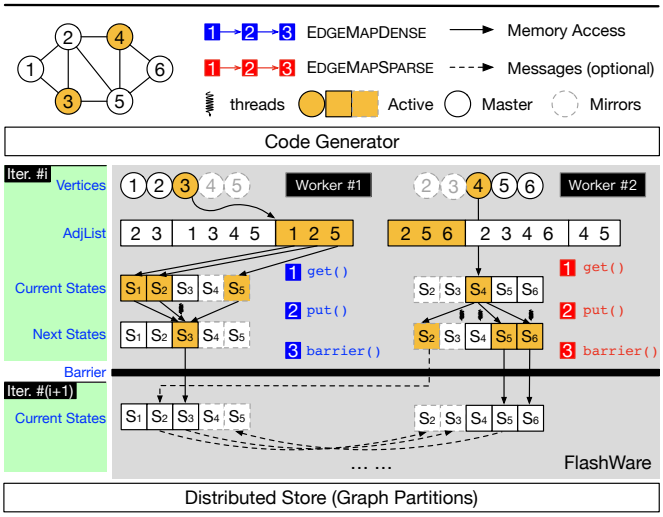

Fig. 2: System architecture and the design of FLASHWARE.

Data access. FLASH's execution is based on the BSP model, hence FLASHWARE distinguishes the *current states* and the *next states*. The current states of a vertex are ensured to be consistent on all workers who access it in the current superstep. On the other hand, the updated values are written in the next states, which is not visible to other vertices in the current superstep and may be different between workers. To save the memory, the next states are only created when necessary.

FLASHWARE exposes two APIs for accessing data:

(1). ***get***$(id : \mathbb{N}) \mapsto vertex$

(2). ***put***$(id : \mathbb{N},$
$v : vertex,$
$R(new : vertex, old : vertex) \mapsto vertex) \mapsto None$

The *get* function is used to access a specific vertex with id through reading the current states. And the *put* function is used to update the vertex with id using a new value v through writing the next states. The data in current states is read-only and consistent during a superstep, thus arbitrary vertices (masters or mirrors) can be obtained safely without data races. As for *put*, the operations on a master are applied immediately, while operations on a mirror may incur concurrent updates (in EDGEMAPSPARSE), so a reduce function R is required to aggregate the new value with the old value. This kind of modification is first applied on the mirror inside this worker, and then further propagated to the corresponding master.

Communication/Synchronization. Both of VERTEXMAP and EDGEMAPDENSE make updates for the masters, thus messages are only from masters to related mirrors. While for EDGEMAPSPARSE, the synchronization procedure is three-phased: a mirror first merges updates to form a temporary new value for this vertex; then mirrors send messages to the master which processes these messages to decide the final state; and

TABLE II: The effect of the optimizations.

Optimization	Effect on performance
O1	reduce the total time from ($\sum T^i_{push}$ or $\sum T^i_{pull}$) to ($\sum min(T^i_{push}, T^i_{pull})$) ideally, T^i is the time for iteration i
O2	reduce T^i from ($T^i_{communication} + T^i_{computation} + T^i_{others}$) to ($T^i_{communication} + T^i_{computation} - T^i_{overlap} + T^i_{others}$)
O3	reduce the size of a single message from (*the total size of all properties*) to (*only that of critical properties*)
O4	reduce the number of messages for synchronizing a single vertex from (*the number of partitions*) to (*the number of necessary mirrors*)

a master broadcasts its final state to necessary mirrors. As a result, there are two rounds of message-passing.

(3). ***barrier***$() \mapsto None$

The interface $barrier()$ acts similar to $MPI\_Barrier$. It is called at the end of a superstep, forcing the workers to wait until all workers have completed the processing for the current superstep. It ensures that all messages are delivered and all updates are applied. When it finishes, the current states and the next states are swapped to make all updates in this superstep to be visible in the current states for starting a new superstep.

Fault Tolerance. Inspired by GraphLab [8], it is possible to achieve fault tolerance for FLASH by constructing the snapshots of FLASHWARE, because the system runs based on the states of FLASHWARE. For constructing a snapshot, the modified data of vertex states since the last snapshot as well as all *vertexSubsets* are saved. Once there is a failure, the system will be recovered from the last checkpoint. We are currently implementing this distributed checkpoint mechanism as an important future work for the production deployment.

C. Code Generation

The code generator generates code to be executed by FLASHWARE, from the high-level user-provided APIs. Take the VERTEXMAP as an illustration, it contains calling $get()$ and $put()$ on the master vertices in parallel inside a worker. Once a master is updated, messages are generated adaptively to synchronize the new state to its mirrors. The $barrier()$ is called at last to ensure that local computation is completed on each worker, and all messages are delivered. After that, the superstep for this VERTEXMAP finishes. The code generation for EDGEMAPDENSE is similar, except that it could call get on mirror vertices, instead of the master vertices only, as Figure 2 shows. the EDGEMAPSPARSE is more complex since it also calls put on mirror vertices.

D. Optimizations

Here we treat the dual update propagation model as the first optimization of FLASH, denoted by O1. On top of O1, we introduce three more system optimizations in FLASH (O2, O3, O4), with the effect of them outlined in Table II. In general, these techniques are useful on all kinds of datasets, except O1 that has no effect on very sparse graphs for which FLASH always uses the sparse mode. As for algorithms, we analyze the algorithms implemented in FLASH for applications of Table V, and report all applicable optimizations in Table V for each algorithm. O1 is applicable for algorithms in which the density (dense or sparse) of the active set changes during computation; otherwise, if the active set is always dense or it is always sparse, it has no effect. The applicability and effect of O2, O3 and O4 are discussed as below.

TABLE III: The rules to decide critical properties.

	VERTEXMAP	EDGEMAPDENSE		EDGEMAPSPARSE	
		source	target	source	target
get	×	✓	×	×	✓
put	×	–	×	–	✓

"✓" means that the property is decided to be critical; "×" means that cannot decide yet; "–" means that this kind of operation never happens.

Overlap communication with computation (O2). For a worker with c cores, FLASH maintains a thread pool containing c threads. Among them, one thread is responsible for inter-worker message sending and one is for receiving via MPI. The other threads perform parallel vertex-centric processing. As separate threads are created to execute message passing, the computation and communication tasks are co-scheduled. By overlapping communication with computation, this technique can speed up all kinds of algorithms, leading to a performance improvement of $4\% \sim 23\%$ (refer to Section V-D and [1]).

Synchronize critical properties only (O3). In some cases, there are multiple vertex properties, but not all of them are *critical*. We say a property critical only if it is accessed by other vertices, thus the update to the master need to be broadcasted to its mirrors. On the contrary, if a property is only read by the master, means that it is only useful in local computation, it is not critical. Given a vertex property, we can decide if it is critical through static analysis of the program. Table III gives the rules based on a classification of the operations on a property. If and only if it is got as the property of the source vertex in an EDGEMAPDENSE function, or it is got/putted as the property of the target in an EDGEMAPSPARSE function, it is critical. For example, the property $v.b$ in MIS (Algorithm 13 in the full version) is an uncritical property, and in GC, the set for recording neighbors' colors is uncritical. For algorithms (e.g., BFS and BC) that do not contain uncritical properties, this technique has no benefits. This optimization reduces the size of a single message from the total size of all properties to only that of critical properties. A quantitative analysis (in the full version) shows that it leads to a reduction on the total communication cost of 79% for MIS and 84% for GC on average.

Communicate with necessary mirrors only (O4). Another intuitive way to eliminate redundant messages is to communicate with only the necessary mirrors. For normal graph applications, the messages are transferred along the edges. Therefore, a master should only communicate with its *necessary* mirrors which locate in a partition containing at least one

neighbor of this vertex. Only in the cases that the programmers define virtual edges for EDGEMAP, which beyond the scope of E (e.g., in CC-log), FLASHWARE synchronizes the update on a master to all partitions, thus this optimization is disabled. For synchronizing a vertex, it reduces the number of messages from the number of partitions to the number of its necessary mirrors. A quantitative analysis on BFS and TC (in [1]) shows that on average, this technique reduces 42% messages for BFS and 35% messages for TC.

V. EVALUATION

In this section, we present the evaluation results to show the superiority of FLASH over previous works in terms of expressiveness, productivity, and efficiency. (1) For expressiveness, we choose 18 representative graph applications from [20] and implement them in FLASH. Each of them represents a general category of algorithms and is frequently used in real-world applications. Some applications are rarely provided in previous works, indicating FLASH's strong expressiveness. (2) For productivity, we try our best to implement algorithms for these applications in state-of-the-art graph frameworks, and show the *Logical Line of Codes* (LLoCs) [27]. Due to the limitations of the expressiveness in other works, we can only implement the sub-optimal version or even failed to implement the naivest version for some algorithms. (3) For efficiency, we compare the execution time of FLASH with other frameworks, which shows that FLASH achieves the best overall performance. We also conduct micro benchmarks to reveal why FLASH runs faster in most of cases.

A. Experimental Setup

We compare FLASH with state-of-the-art graph processing frameworks using following setups:

Platform. All our experiments are conducted on a 20-node Intel(R) Xeon(R) CPU E5-2682 v4 based system. Each node has 32 cores running at 2.50 GHz and 512GB of main memory. All nodes are connected with a 10Gb ethernet. In order to perform the comparison, the latest version of Pregel+, PowerGraph, Gemini and Ligra are installed on the cluster. We leverage OpenMP to manage threads and Open MPI to manage message passing between nodes.

Datasets. We use a collection of real-world graphs, including social networks (SN), web graphs (WG), and road networks (RN), with their basic characteristics illustrated in Table IV. For the social network, it is characterized by a very skew distribution of edges, usually with some "hot" vertices having an extremely high degree. For the road network, it has a very long diameter and fewer associated edges per vertex. And the web graph lies somewhere in middle. For unweighted graphs, random weights are added to each edge if necessary.

Applications. We choose 18 representative graph applications, as shown in Table V. CC and BFS are well supported by all tested frameworks, because the ISVP algorithms perform well enough for most cases. We also implemented an optimized CC algorithm [19] in FLASH since it performs better on large-diameter graphs. For BC, MIS and MM, every system we evaluated is able to express a basic algorithm correctly, but they failed to express the advanced versions, suffering either poor performance or complicated programs. FLASH is not only able to implement the basic version with less effort, but also make the advanced version possible. For KC, TC and GC, even the naivest algorithms are not feasible to implement in some frameworks, while FLASH is able to implement them with less effort, because of the support for non-ISVP algorithms. As for SCC, BCC, LPA, MSF, they are rarely provided by existing graph frameworks, and it is often failed to implement them. While it is easy for FLASH to do so. The last six applications result in local algorithms, in which a piece of message will be propagated within only several hops. Since this kind of workloads are not the main target of general graph processing frameworks, few of them are provided by the competitors, which verified the expressiveness of FLASH once again.

TABLE IV: A collection of real-world graphs.

Abbr.	Dataset	|V|	|E|	Diameter	Domain
OR [37]	soc-orkut	3.07M	117M	9	SN
TW [38]	soc-twitter	41.7M	1.47B	15	SN
US [39]	road-USA	23.9M	28.9M	1452	RN
EU [39]	europe-osm	50.9M	54.1M	2037	RN
UK [39]	uk-2002	18.5M	298M	25	WG
SK [39]	sk-2005	50.6M	1.95B	23	WG

TABLE V: A collection of representative graph applications.

Abbr.	Application	Abbr.	Application
CC[123]	connected components	BFS[124]	breadth-first search
BC[124]	betweenness centrality	MIS[1234]	maximal independent set
MM[124]	maximal matching	KC[234]	k-core decomposition
TC[234]	triangle counting	GC[234]	graph coloring
SCC[124]	strongly CC	BCC[12]	biconnected components
LPA[234]	label propagation	MSF[2]	minimum spanning forest
RC[234]	rectangle counting	CL[234]	k-clique counting
3PC[234]	3-path counting	DC[234]	diamond counting
TTC[234]	tailed-triangle counting	KCS[24]	k-core searching

The four optimizations in FLASH: (O1) dual update propagation[1], (O2) overlap[2], (O3) critical properties[3] and (O4) necessary mirrors[4] optimization.

Baselines. Four representative state-of-the-art graph processing frameworks are tested as the baselines: Pregel+, PowerGraph, Gemini and Ligra. Pregel+ is a framework that uses Pregel's vertex-centric model. Compared with other Pregel-like frameworks (e.g., Giraph [7] and GPS [12]), Pregel+ provides simpler interface and higher efficiency. PowerGraph is the representative framework that adopts the GAS model. Gemini is a state-of-the-art framework which is reported to significantly outperform all well-known existing distributed graph processing frameworks, however, its expressiveness is weaker than others'. Ligra is a shared-memory framework which provides similar interfaces with FLASH.

These frameworks provide some pre-optimized built-in algorithms, but none of them implements for all applications in Table V, due to the limitations of expressiveness. For fair comparison, we test the built-in algorithms if provided (if multiple implementations provided, choose the fastest one), or, we try our best to implement them in these frameworks.

TABLE VI: Execution time for the first eight applications on six datasets (in seconds).

App.	Data	Pregel+	PowerG.	Gemini	Ligra	FLASH	App.	Data	Pregel+	PowerG.	Gemini	Ligra	FLASH
CC	OR	9.21	5.31	1.24	0.49	**0.48**	BFS	OR	3.07	6.27	0.87	**0.35**	**0.35**
	TW	99.31	281.93	8.60	10.09	**6.38**		TW	31.47	48.11	4.61	**2.28**	6.16
	US	435.42	1832.2	524.34	323.43	**30.96**		US	202.79	1512.3	519.01	244.01	**12.17**
	EU	1740.0	6749.7	1302.3	663.10	**76.47**		EU	1035.5	4453.4	1445.4	506.72	**50.32**
	UK	33.56	26.33	3.33	**2.09**	2.51		UK	5.94	15.51	2.78	**1.09**	2.26
	SK	132.97	307.30	5.57	**4.07**	7.02		SK	29.33	35.96	3.53	**1.92**	6.02
BC	OR	11.23	13.40	1.73	0.81	**0.54**	MIS	OR	11.22	12.30	1.78	2.66	**0.51**
	TW	110.29	121.71	**8.15**	21.62	11.77		TW	55.62	176.77	4.66	20.61	**4.58**
	US	516.86	3066.8	1007.1	411.25	**16.94**		US	4.55	22.58	3.93	1.10	**0.94**
	EU	2981.1	OT	2861.8	978.21	**129.64**		EU	254.88	722.41	188.22	122.41	**12.14**
	UK	22.61	39.91	6.24	**2.18**	3.87		UK	14.05	65.64	20.46	4.92	**1.83**
	SK	116.13	127.23	7.54	**7.08**	11.49		SK	77.54	108.54	13.37	9.24	**5.13**
MM	OR	OT	OT	497.15	889.61	**22.27**	KC	OR	678.44	1140.6	–	302.65	**4.03**
	TW	OT	OT	OT	OT	**25.15**		TW	4937.4	OT	–	1313.4	**29.26**
	US	13.00	65.66	6.96	3.69	**3.03**		US	232.18	68.80	–	16.11	**2.12**
	EU	428.87	1547.7	253.25	182.36	**19.17**		EU	OT	634.68	–	195.04	**10.44**
	UK	OT	OT	1091.8	518.83	**22.11**		UK	2924.6	2682.4	–	577.72	**5.38**
	SK	OT	OT	OT	OT	**114.76**		SK	OT	OT	–	3702.8	**44.16**
TC	OR	529.61	27.86	–	12.90	**3.32**	GC	OR	OT	13.26	–	–	**9.72**
	TW	OOM	720.01	–	OT	**49.10**		TW	OT	426.37	–	–	**264.44**
	US	17.90	6.48	–	**0.57**	1.09		US	10.29	13.11	–	–	**2.38**
	EU	32.56	10.91	–	**0.53**	2.29		EU	242.59	**43.81**	–	–	54.61
	UK	OOM	17.44	–	14.23	**7.00**		UK	2219.7	36.19	–	–	**35.67**
	SK	OOM	211.67	–	OT	**70.59**		SK	OT	706.21	–	–	**331.72**

"–" means that we fail to implement an available algorithm for this case because of the limitations in expressiveness; "OOM" means that the tested algorithm failed due to exhausted memory. "OT" means that the execution did not terminate within 5000s.

TABLE VII: Execution time for four more complex applications on six datasets (in seconds).

App.	Data	Baseline	FLASH	App.	Data	Baseline	FLASH
SCC	OR	120.76	**1.24**	BCC	OR	303.93	**5.57**
	TW	949.60	**13.80**		TW	3615.0	**75.85**
	US	719.91	**57.84**		US	3844.7	**169.58**
	EU	3021.1	**161.35**		EU	OT	**486.14**
	UK	223.22	**5.55**		UK	879.91	**22.82**
	SK	1335.5	**18.26**		SK	2991.8	**55.20**
LPA	OR	155.90	**16.83**	MSF	OR	55.96	**6.96**
	TW	1433.9	**100.31**		TW	867.54	**72.51**
	US	49.11	**2.77**		US	**25.42**	29.96
	EU	276.20	**25.57**		EU	**64.86**	68.66
	UK	299.62	**11.06**		UK	55.25	**29.74**
	SK	OT	**78.25**		SK	477.72	**86.84**

The baseline results for SCC, BCC and MSF are tested on Pregel+, and the baseline results for LPA are tested on PowerGraph.

TABLE VIII: Execution time of FLASH for six local applications on six datasets (in seconds).

Data	RC	CL	3PC	DC	TTC	KCS
OR	12.49	20.33	12.51	31.50	12.12	2.03
TW	140.16	OT	OOM	OOM	OOM	7.45
US	1.31	1.22	1.29	1.93	1.40	2.22
EU	2.75	2.39	2.46	3.24	2.48	2.20
UK	14.65	420.12	27.09	77.23	27.20	2.21
SK	176.78	OT	OOM	OOM	OOM	18.17

For KCS, $k = 3$ on US and EU and $k = 50$ on others. As for CL, $k = 4$.

B. Overall Performance

Table VI reports the results on overall execution time of the first eight applications on six datasets. All these tests are conducted on 4 nodes of the cluster except Ligra, which only uses a single node. For fair comparison, the initial preprocessing time (for data loading and partitioning) and postprocessing time (e.g., writing the results) of every framework are not recorded. Specially, Pregel+ may decompose the algorithm (*e.g.,* BC, SCC, and BCC) into several individual subalgorithms and chain them by taking the output of the previous as the input of the next. In this situation, the data sharing time among sub-algorithms will be recorded. Table VII reports the results of the four more complicated applications. Since they are difficult or not supported to be implemented in previous works, we only compare the performance of FLASH with the most efficient implementation provided by other frameworks.

In 95.2% cases, FLASH provides competitive performance compared with the one that performs best (within $2\times$ slowdown); and in 84.5% cases, it is faster than all other compared frameworks, with the speedup up to 2 orders of magnitude. For example, when calculating MM on the TW dataset, FLASH only takes 25.15 seconds, while all the other frameworks failed to get the results within 5000 seconds. Another example is SCC, which requires complex computation and is only provided by Pregel+ as far as we know. The implementation in Pregel+ is $22.7\times$ to $54.6\times$ slower than FLASH. PowerGraph performs efficiently on GC since it implements an asynchronous algorithm, which converges faster than a BSP-based algorithm. Ligra is faster than FLASH in some cases because it is a shared-memory system, with the communication cost much cheaper than that of distributed systems.

To further validate FLASH's expressiveness, some local algorithms are evaluated. Besides TC, which is provided by

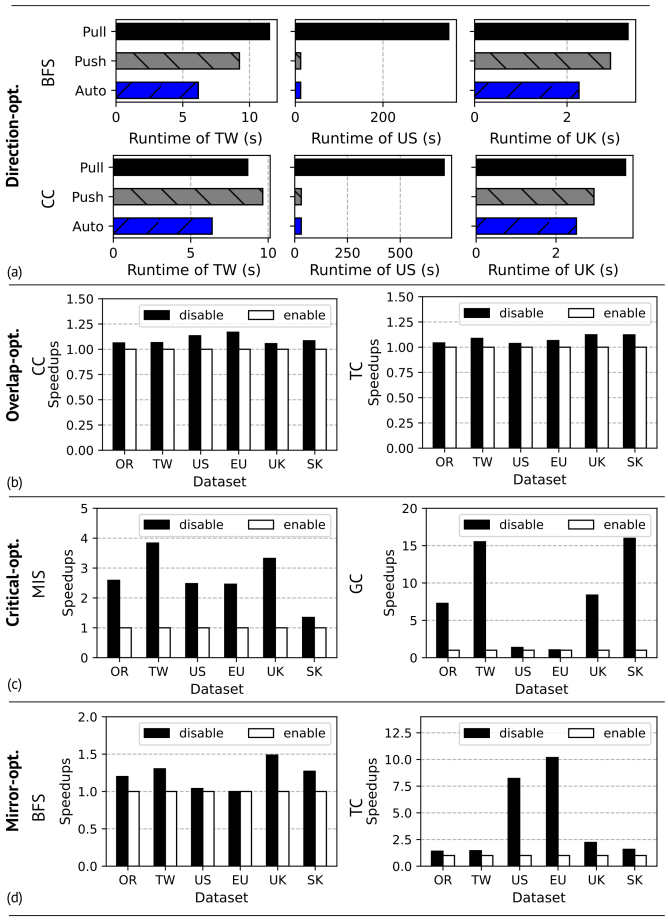

Fig. 3: The effectiveness of (a) dual-direction, (b) overlap, (c) critical properties and (d) necessary mirrors optimization.

some compared frameworks, we implemented algorithms in FLASH for matching other subgraphs including rectangles, 3-paths, tailed triangles and diamonds. Searching for k-cliques and k-cores are also implemented in FLASH without much effort. Table VIII reports the time performance. Since this kind of local algorithms are often non-ISVP, other general graph processing frameworks mainly focus on global algorithms, while FLASH is suitable for both global and local algorithms.

C. Productivity

To demonstrate the productivity of FLASH, the LLoCs are counted, as shown in Table I. Note that we only consider LLoCs in the core functions, while ignoring the comments, input/output expressions, and data structure (e.g., the graph) definitions. Gemini fails to express most algorithms since its programming model is most limited. PowerGraph needs lots of code for TC since it does not provide the serialization/de-serialization semantics for users to exchange neighbor-lists. Although Pregel+ is able to express some complex non-ISVP algorithms (e.g., SCC and BCC), it is usually intractable. In fact, its algorithms for these applications are decomposed into several parts, with each part constitutes of an individual sub-algorithm and needs all necessary functions to be programmed. This is obviously not friendly to the users. Moreover, since every sub-algorithm needs its own implementation for parsing the input from the previous sub-algorithm and outputting data for the next one (which are not counted in the LLoCs), the actual lines of code are further more than the results in Table I (811 lines in total for SCC and 3017 lines for BCC). This algorithm decomposition also results in poor performance.

Although all compared frameworks evaluated are claimed to provide succinct interface, and have been widely used, FLASH requires less effort in implementation than other works in all tested cases, showing that FLASH achieves the best productivity when expressing both the ISVP and non-ISVP algorithms. We provide more examples in the full version [1] for the readers to further judge the succinctness and readability.

D. Micro Benchmarks

To further evaluate how the optimization techniques impact FLASH's performance, we conduct several micro benchmarks. This section summarizes the results, please refer to the full version for more detailed experimental data.

Dual update propagation model. Adaptive switching between the *pull* (dense) mode and the *push* (sparse) mode according to the density improves the performance of FLASH significantly. For algorithms such as BFS and BC, in common cases, the active set is initially sparse, switches to dense after a few iterations and then switches back to sparse later. For another category of algorithms (e.g., CC, MM, MIS and GC), the active set starts as dense, and becomes sparser as the algorithm continues. For TC and PageRank, all vertices are active in each step (i.e., in the dense mode all the time), so this adaptive switching is disabled. We demonstrate the effectiveness of adaptive switching by compare its performance with using either *push* or *pull*. Figure 3 (a) shows the execution time of BFS and CC, as expected, the performance gap is quite significant. In conclusion, the dual update propagation scheme always achieves the best performance, which outperforms the push-only and pull-only alternatives by $1.0\times \sim 1.5\times$. The US graph is sparse with a very low average degree, thus our adaptive switching falls into the sparse mode all the time, while the dense mode consumes much longer execution time.

Overlap communication with computation. We compare the execution time of two algorithms with and without this optimization, as shown in Figure 3 (b). According to our evaluation, this optimization leads to a performance improvement of $4\% \sim 23\%$. It is applicable for all algorithms, and it is typically more effective for algorithms that the computation is lightweight and the communication time is comparative with that of computation, such as CC and MM.

Synchronize critical properties only. As analyzed in Section IV-D, this strategy is useful when there are uncritical properties. Figure 3 (c) shows the execution time of MIS and GC on different graphs when enabling and disabling this optimization, showing that it could lead to a considerable speedup. These two algorithms represent different workloads. In MIS, the uncritical property is a 1-byte boolean variable, while in GC, the uncritical property is the neighbor set which can be very large. A more detailed analysis on the

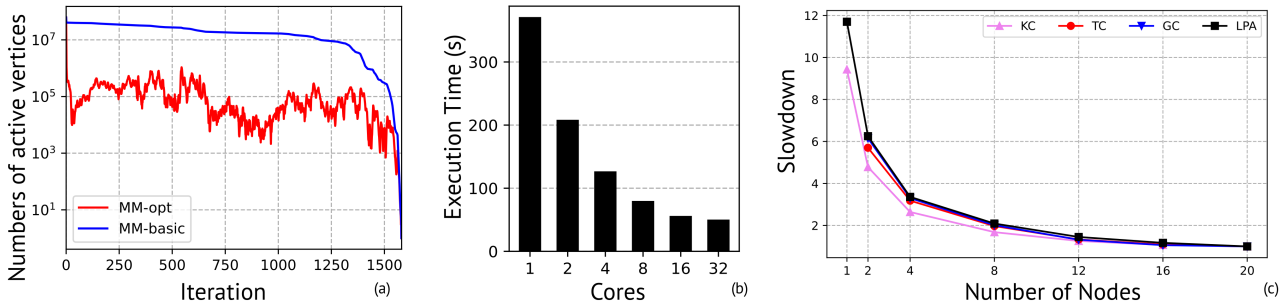

Fig. 4: (a) The number of active vertices for MM-basic and MM-opt on TW. (b) Performance of TC on TW with varying cores per node (4 nodes in total). (c) Performance of KC, TC, GC and LPA on TW with varying nodes (4 cores per node).

communication cost (by counting the exact amount of network traffic) shows that the average reduction on communication is 79% for MIS and 84% for GC. The average speedup of MIS and GC are 2.7× and 8.3×, respectively.

Communicate with necessary mirrors only. To evaluate the effect of communicating with only necessary mirrors, we report the execution time of BFS and TC when enabling and disabling this optimization (Figure 3 (d)). BFS contain many rounds of small messages, while in TC, the messages can be very large. This technique reduces the number of messages by 13% ∼ 79% for BFS and 6% ∼ 85% for TC, thus improves the performance by 1.2× for BFS and 4.2× for TC on average.

E. Advanced Implementations

The high expressiveness of FLASH does not only allow users to implement algorithms with less effort, but also allows users to implement the advanced version of some algorithms for higher performance. Consider the MM application as an example, we implement a basic algorithm in FLASH (MM-basic), as well as an optimized one (MM-opt), as described in the full version. Other frameworks cannot implement MM-opt since they do not support to define arbitrary edges sets. The advanced algorithm has higher efficiency than the basic one, which means it will touch less vertices and edges during the execution. Figure 4 (a) compares the number of active vertices (size of the frontier) in all iterations for both algorithms on the TW dataset. The significant reduction in active vertices leads to a considerable speedup of 70.1× (1763.0s for MM-basic and 25.15s for MM-opt). This is also the main reason that FLASH significantly outperforms other works. Another example is the application CC, for which the CC-basic algorithm takes 169.6s on US, and the CC-opt only takes 30.96s. For applications that FLASH implements the same algorithm as other systems such as BFS and TC, it provides a comparative performance, as shown by Table VI.

F. Scalability

We examine the scalability of FLASH in terms of both intra-node and inter-node. We first compare the performance of FLASH while varying the cores of each node as 1, 2, 4, 8, 16, 32. Figure 4 (b) presents the execution time of running TC on TW using 4 nodes, which shows a reasonable trend of scalability, achieving speedup of 1.8×, 2.9×, 4.7×, 6.7× and 7.5× at 2, 4, 8, 16, 32 cores, respectively. The other cases show similar trends, except the cases that the communication time dominates the execution such as running BFS on the US graph, on which the scalability is poor for all frameworks. The reduction on execution time after 4 cores slows down since when more cores are used, the scheduling cost and memory contention inside a node increase.

We also conduct experiments to evaluate the inter-node scalability using up to 20 nodes, as shown by Figure 4 (c). All execution time are normalized to slowdown of the best case in question. When increasing the cluster size from 1 node to 20 nodes, the speedup is 11.7× for LPA and 9.4× for KC (TC and GC failed with 1 node). The reduction on execution time becomes slower with the number of nodes increases, because the computation time reduces but the communication cost may increase. This is also the common pattern that limits the scalability of all kinds of distributed frameworks. A piecewise breakdown analysis on the execution time validates this, showing that with the increase of the cluster size, the computation time decreases nearly linearly, while the communication time accounts to more and more percentages of the total time. For example, the computation time for TC with 4 nodes is 3.6× of that with 16 nodes, while the communication time is 1.2×.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we present FLASH, a framework for programming distributed graph algorithms. We track three essential metrics for graph frameworks from exploring more advanced and complex graph algorithms: expressiveness, productivity and efficiency. Providing a new high-level programming model and an efficient system implementation, FLASH leads to easy programming and outstanding performance for a wide variety of graph algorithms. In addition to the algorithms we discussed, we believe that other algorithms can also benefit from our framework, since the huge potentials it revealed.

As a topic for future work, FLASH is proposed to be a component of GraphScope (the unified framework for large-scale graph processing at Alibaba) for easing the programming. Meanwhile, GRAPE [16] is the current component for executing graph algorithms in GraphScope. While GRAPE focuses on the low-level execution of graph algorithms, FLASH emphasizes their easy programming in the higher level. Their technique stacks are thus orthogonal in that a FLASH program can be compiled into the GRAPE runtime to combine their advantages, which will also be our future proposal.

REFERENCES

[1] (2022) Flash: A framework for programming distributed graph processing algorithms. [Online]. Available: https://graphscope.io/flash-full.pdf

[2] S. Brin and L. Page, "Reprint of: The anatomy of a large-scale hypertextual web search engine," *Computer networks*, vol. 56, no. 18, pp. 3825–3833, 2012.

[3] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang, "A distributed graph engine for web scale rdf data," *Proc. VLDB Endow.*, vol. 6, no. 4, p. 265–276, Feb. 2013.

[4] J. Domke, T. Hoefler, and W. E. Nagel, "Deadlock-free oblivious routing for arbitrary topologies," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 616–627.

[5] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick, "Parallel de bruijn graph construction and traversal for de novo genome assembly," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 437–448.

[6] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *ACM SIGMOD International Conference on Management of Data*, 2010, pp. 135–146.

[7] C. Avery, "Giraph: Large-scale graph processing infrastructure on hadoop," in *Proceedings of the Hadoop Summit*, vol. 11, no. 3, 2011, pp. 5–9.

[8] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning in the cloud," *arXiv preprint arXiv:1204.6078*, 2012.

[9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs." in *OSDI*, vol. 12, no. 1, 2012, p. 2.

[10] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th USENIX symposium on operating systems design and implementation (OSDI 14)*, 2014, pp. 599–613.

[11] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 301–316.

[12] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proceedings of the 25th international conference on scientific and statistical database management*, 2013, pp. 1–12.

[13] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Effective techniques for message reduction and load balancing in distributed graph computation," in *Proceedings of the 24th International Conference on World Wide Web*, 2015, pp. 1307–1317.

[14] R. Chen, X. Ding, P. Wang, H. Chen, B. Zang, and H. Guan, "Computation and communication efficient graph processing with distributed immutable view," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, 2014, pp. 215–226.

[15] T. Li, C. Ma, J. Li, X. Zhou, K. Wang, D. Zhao, I. Sadooghi, and I. Raicu, "Graph/z: A key-value store based scalable graph processing system," in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 516–517.

[16] W. Fan, W. Yu, J. Xu, J. Zhou, X. Luo, Q. Yin, P. Lu, Y. Cao, and R. Xu, "Parallelizing sequential graph computations," *ACM Transactions on Database Systems (TODS)*, vol. 43, no. 4, pp. 1–39, 2018.

[17] V. Kalavri, V. Vlassov, and S. Haridi, "High-level programming abstractions for distributed graph processing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 2, pp. 305–324, 2017.

[18] P. Stutz, A. Bernstein, and W. Cohen, "Signal/collect: graph algorithms for the (semantic) web," in *International Semantic Web Conference*. Springer, 2010, pp. 764–780.

[19] L. Qin, J. X. Yu, L. Chang, H. Cheng, C. Zhang, and X. Lin, "Scalable big graph processing in mapreduce," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 827–838.

[20] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11 – 15.

[21] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 135–146.

[22] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "Graphit: A high-performance graph dsl," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–30, 2018.

[23] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of mathematical sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[24] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo, "K-core decomposition of large networks on a single pc," *Proceedings of the VLDB Endowment*, vol. 9, no. 1, pp. 13–23, 2015.

[25] L. Yuan, L. Qin, X. Lin, L. Chang, and W. Zhang, "Effective and efficient dynamic graph coloring," *Proceedings of the VLDB Endowment*, vol. 11, no. 3, pp. 338–351, 2017.

[26] J. Shi, L. Dhulipala, and J. Shun, "Parallel clique counting and peeling algorithms," 2020. [Online]. Available: https://arxiv.org/abs/2002.10047

[27] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. Boehm, "A sloc counting standard," in *Cocomo ii forum*, vol. 2007. Citeseer, 2007, pp. 1–16.

[28] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, p. 103–111, aug 1990. [Online]. Available: https://doi.org/10.1145/79173.79181

[29] S. Salihoglu and J. Widom, "Optimizing graph algorithms on pregel-like systems," *Proceedings of the VLDB Endowment*, vol. 7, no. 7, 2014.

[30] S. Heidari, Y. Simmhan, R. N. Calheiros, and R. Buyya, "Scalable graph processing frameworks: A taxonomy and open challenges," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–53, 2018.

[31] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, 2010, pp. 303–314.

[32] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, "Brief announcement: the problem based benchmark suite," in *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, 2012, pp. 68–70.

[33] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, pp. 35–41, 1977.

[34] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, 2013, pp. 456–471.

[35] K. Zhang, R. Chen, and H. Chen, "Numa-aware graph-structured analytics," in *Proceedings of the 20th ACM SIGPLAN symposium on principles and practice of parallel programming*, 2015, pp. 183–193.

[36] G. Karypis and V. Kumar, "Metis – unstructured graph partitioning and sparse matrix ordering system, version 2.0," 01 1995.

[37] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowledge and Information Systems*, vol. 42, no. 1, pp. 181–213, 2015.

[38] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th international conference on World wide web*, 2010, pp. 591–600.

[39] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: https://networkrepository.com